DOMÄNENMODELLIERUNG MIT F#

Die bessere Wahl!

Warum F# besser ist als C#, dargestellt am Beispiel der Domain Financial Contracts.

F# ist die Programmiersprache der Wahl auf der .NET-Plattform und in allen wichtigen Belangen C# überlegen. Dafür gibt es viele Gründe: Weil die Abstraktionsfähigkeiten von F# mächtiger sind und die Notation kompakter, sind F#-Programme kürzer und damit schneller zu entwickeln. Das Typsystem unterstützt durch Typinferenz und parametrische Polymorphie die Programmierung deutlich aktiver als C#. Funktionale Programmierung und funktionale Datenstrukturen senken die Kopplung in der Architektur und reduzieren damit die Fehlerquote erheblich. Die Sprache ist deutlich stabiler als C#, mit nur wenigen substanziellen Änderungen seit der Einführung 2005 und der Auslieferung als fester Bestandteil von Visual Studio im Jahr 2010.

F# ist außerdem eine deutlich kleinere und gleichzeitig weniger komplexe Sprache als C# und lässt sich deshalb auch leichter erlernen. Insofern C# die besonders mächtigen Features aus F# ebenfalls unterstützt, sind diese meist nachträglich von dort oder aus anderen funktionalen Sprachen importiert worden.

Es gibt also viele technische Gründe, um F# gegenüber C# vorzuziehen. Sie vermitteln aber noch keinen guten Eindruck davon, was all diese Technik eigentlich für praktische Veränderungen im täglichen Entwicklungsprozess möglich macht. Um einen Aspekt davon geht es in diesem Artikel, nämlich um die Domänenmodellierung [1]. Als Beispiel dafür dient ein Klassiker der funktionalen Programmierung, nämlich die Arbeit an Financial Contracts [2], deren Grundlagen hier in F# nachvollzogen werden sollen. Vorkenntnisse in F# sind für das Verständnis dieses Artikels nicht notwendig, alle benutzten F#-Konstrukte werden erläutert.

Financial Contracts

Bei der Arbeit an Financial Contracts geht es um komplexe Finanzderivate. So ein Finanzderivat ist ein Vertrag zwischen zwei Parteien (darum *Contracts*), bei dem nach bestimmten Regeln zu bestimmten Zeitpunkten Zahlungen zwischen den Parteien festgelegt werden.

Um eine solche Domäne zu modellieren, ist es sinnvoll, mit Bankerinnen oder Bankern zu sprechen und sie zu fragen: „Was ist das einfachste Finanzderivat, das du kennst?“ Viele würden „Zero-Bonds“ antworten, auf Englisch „zero-coupon bonds“. Der Name bezieht sich darauf, dass die Verträge keine Zinsen enthalten. So etwas könnte wie folgt aussehen:

- Bekomme 100 Euro am 24.12.2020.
- Zahle 100 britische Pfund (GBP) am 24.12.2021.

Als Nächstes könnte man das in einem Datentyp abbilden. Dafür braucht man so etwas wie eine Klasse *Zcb* mit Attributen für Datum, Betrag und Währung. In F# könnte das beispielsweise so aussehen:

```
type Direction = Long | Short
type Amount = double
type Currency = EUR | GBP
type Date = Date of string
type Contract = Zcb of Date * Amount * Currency
```

Zunächst werden Typen für die drei Attribute eines Zero-Bonds definiert – allesamt grobe Vereinfachungen, damit wir uns auf das Wesentliche konzentrieren können: Der Typ *Direction* sagt, in welche Richtung die Zahlungen fließen – *Long* für „ich bekomme“ und *Short* für „ich zahle“. Den senkrechten Strich dazwischen kann man als „oder“ lesen. *Amount* für den Betrag ist ein Synonym für *double*, an Währungen sind nur Euro und britisches Pfund zugelassen.

Für das Datum genügt eine Zeichenkette wie *2020-12-24*, mit einem Wrapper namens *Date*. Für den Vertrag selbst schließlich definiert die Typdefinition von *Contract* einen Konstruktor *Zcb* mit genau diesen drei Attributen. Das Sternchen zwischen den Attributen steht für „kartesisches Produkt“, ist also das „und“ zu dem „oder“, für das der senkrechte Strich steht.

Diese wenigen Zeilen genügen schon, um die beiden Beispielverträge als Daten zu repräsentieren:

```
let zcb1 =
  Zcb (Long, Date "2020-12-24", 100.0, EUR)
let zcb2 =
  Zcb (Short, Date "2021-12-24", 100.0, GBP)
```

Das *let* bindet eine Variable – danach kann man im Code also *zcb1* und *zcb2* verwenden. Der Konstruktor *Zcb* wird automatisch vom F#-Compiler generiert.

Der F#-Code ist also kürzer und kommuniziert in ein paar Zeilen präzise, wie das Domänenmodell aussieht. Das an sich ist schon praktisch. Viel wichtiger aber ist, dass die kompakte Notation erlaubt, effektiver über das Domänenmodell nachzudenken.

Wenn wir als Entwickler mit den Domänenexperten sprechen, erwähnen diese unter Umständen, dass es neben den Zero-Bonds auch Swaps gibt, bei denen (etwas vereinfacht) zwei Zahlungen kombiniert werden – zum Beispiel eine über 100 Euro in die eine Richtung, die andere über 100 britische Pfund in die andere Richtung. Natürlich könnten man dafür einen weiteren Konstruktor zu *Contract* hinzufügen, beispielsweise so:

```
type Contract =
  | Zcb of Date * Amount * Currency
  | Swap of Date * Amount * Currency * Amount * Currency
```

Das wäre die direkte Modellierung des Swaps. Aber da beschleicht einige vielleicht das Gefühl, es könnte besser sein, einen Swap als Kombination von zwei Zero-Bonds darzustellen. Dieser Gedanke ebnet den Weg für eine grundsätzliche Idee, nämlich nicht jeden Vertrag für sich zu definieren, sondern stattdessen einen Vertragsbaukasten anzulegen, aus dem die real existierenden Verträge zusammengesetzt werden können – und zwar nicht nur die Verträge, die es schon gibt, sondern auch zukünftige Verträge.

Damit diese Idee funktioniert, braucht es möglichst universell verwendbare Bauteile – wie in einem Legobaukasten sind das kleine, regelmäßig geformte Teile, nicht die Spezialanfertigungen aus der Technic-Reihe.

In einem Domänenmodell entwickelt man solche Bauteile am besten, indem man die bestehenden Domänenobjekte in möglichst kleine Teile zerlegt. Den Anfang machen die Zero-Bonds. Auch wenn Banker sich kaum einen einfacheren Vertrag vorstellen können, hat dieser drei separate Bestandteile:

- Datum,
- Betrag,
- Währung.

Aufgrund dieser Einsicht wird nun der *Contract*-Typ überarbeitet, diesmal mit drei Konstruktoren:

```
type Contract =
  | Later of ...
  | Multiple of ...
  | One of Currency
```

Noch sind die Attribute der ersten beiden Konstruktoren offengelassen – sie kommen gleich an die Reihe. Zuerst geht es um den einfachsten Fall, nämlich *One* – er liefert eine Einheit einer bestimmten Währung, also zum Beispiel 1 Euro oder 1 GBP. Erweitert man den Typ *Currency*, können das auch andere Währungen sein, oder eine bestimmte Aktie oder eine Kuh – aber eben immer nur eine.

One reicht natürlich nicht, wenn 100 Euro repräsentiert werden sollen – dafür ist der Konstruktor *Multiple* da. Man ist versucht, da einfach Folgendes hinzuschreiben:

```
Multiple of Amount * Currency
```

Damit ließen sich 100 Euro als *Multiple (100.0, EUR)* hinschreiben, aber gleichzeitig wäre der *One*-Konstruktor redundant und damit hinfällig – für einen Euro würde folglich *Multiple (1.0, EUR)* reichen.

Stattdessen hilft hier ein Designprinzip aus der funktionalen Programmierung, das man als „Finde den Kombinator!" beschreiben könnte. Ein Kombinator ist eine Operation, die aus einem Domänenobjekt eines Typs ein etwas „größeres" Domänenobjekt des gleichen Typs macht. Entsprechend dieses Prinzips sollten wir deshalb anstreben, dass *Multiple* in der Lage ist, aus einem Vertrag einen „größeren Vertrag" zu machen. Das geht, indem *Multiple* nicht auf *Currency* beschränkt, sondern auf *Contract* ausgeweitet wird:

```
Multiple of Amount * Contract
```

Damit lassen sich 100 Euro als *Multiple (100.0, One EUR)* ausdrücken, aber auch andere Verträge „vervielfachen". Das Kombinator-Prinzip findet sich übrigens etwas verklausuliert auch im Domain-Driven Design wieder:

„Where it fits, define an operation whose return type is the same as the type of its argument(s)." – Eric Evans, Domain-Driven Design [3].

Es fehlen noch die Attribute des *Later*-Vertrags: Auch dieser Konstruktor lässt sich als Kombinator formulieren, was dann vorläufig zu folgendem Datentyp führt:

```
type Contract =
  | One of Currency
  | Multiple of Amount * Contract
  | Later of Date * Contract
```

Während also die *One*- und *Multiple*-Verträge als Ausführungszeitpunkt immer „jetzt" haben, besagt *Later (date, contract)*, dass jetzt der Vertrag geschlossen wird, zum Zeitpunkt *date* den Vertrag *contract* abzuschließen. Das ermöglicht es nun, *zcb1* und *zcb2* mithilfe dieser drei Konstruktoren hinzuschreiben:

```
let zcb1 =
  Later (Date "2020-12-24",
  Multiple (100.0, One EUR))

let zcb2 =
  Later (Date "2021-12-24",
  Multiple (100.0, One GBP))
```

Besser sogar: Wenn geplant ist, noch viele weitere Zero-Bonds auszustellen, kann man eine Funktion definieren, die das direkt erledigt:

```
let zcb (date: Date) (amount: Amount) (
  currency: Currency): Contract =
  Later (date, Multiple (amount, One currency))

// Damit gehen zcb1 und zcb2 so:

let zcb1 = zcb (Date "2020-12-24") 100.0 EUR
let zcb2 = zcb (Date "2021-12-24") 100.0 GBP
```

Das sieht fast so aus wie bei der Verwendung des ursprünglichen *Zcb*-Konstruktors. Es gibt aber zwei augenfällige Unterschiede:

- Die Funktion *zcb* fängt mit einem Kleinbuchstaben an, die Konstruktoren sind großgeschrieben.
- Beim Konstruktor *Zcb* sind die Argumente durch Kommata getrennt, bei der Funktion *zcb* durch Leerzeichen. ▶

Letzteres hat einen tieferen, technischen Grund: F# kennt – anders als C# – nur einstellige Funktionen, also Funktionen mit einer Ein- und einer Ausgabe. Diese Uniformität macht vieles einfacher, wirft aber die Frage auf, wie denn Funktionen mehrere Eingaben akzeptieren können. Dafür gibt es in F# zwei Methoden:

- Mehrere Argumente werden in ein kartesisches Produkt (kurz Tupel) verpackt, indem Klammern drumherum und Kommata dazwischenkommen. So ist es immer bei Konstruktoren wie *Zcb*.
- Die Argumente werden nacheinander übergeben: Das heißt, die Funktion akzeptiert das erste Argument und retourniert dann eine weitere Funktion, welche das nächste Argument akzeptiert und so weiter: Das geht einfach durch Leerzeichen zwischen den Argumenten. So ist es bei normalen Funktionen wie *zcb*.

Das heißt, obwohl *Zcb (Long, Date "2020-12-24", 100.0, EUR)* so aussieht, als würde *Zcb* mit drei Argumenten aufgerufen – wie es in C# der Fall wäre –, ist es in F# nur ein Argument, und zwar ein Drei-Tupel. Da Funktionsaufrufe in F# in der Regel Funktion und Argument durch Leerzeichen trennen, findet sich hier ebenfalls ein Leerzeichen zwischen *Zcb* und der öffnenden Klammer.

Damit alles diesbezüglich ein bisschen uniformer wird, empfiehlt es sich, Funktionen zu definieren, welche die Konstruktoren wrappen:

```
let one currency = One currency
let multiple amount contract =
  Multiple (amount, contract)
let later date contract = Later (date, contract)
```

Man sieht hier übrigens, dass man die Typen gar nicht hinschreiben muss – der Compiler findet sie automatisch heraus. Es ist oft nützlich, sie trotzdem hinzuschreiben, um die Lesbarkeit zu erhöhen, hier aber sind sie offensichtlich. Die Wrapper werden sich später noch aus einem anderen Grund als nützlich erweisen. Sie lassen sich auch in der Definition von *zcb* benutzen:

```
let zcb (date: Date) (amount: Amount)
  (currency: Currency): Contract =
  later date (multiple amount (one currency))
```

Diese Definition zeigt, dass nun die „eingebauten" Kombinatoren wie *later* und *multiple* in Form und Verwendung den „abgeleiteten" Kombinatoren wie *zcb* gleichen. Wir können also eine Bibliothek von Funktionen für die Konstruktion von Verträgen entwickeln, der man von außen gar nicht ansieht, was eingebaut ist und was abgeleitet. Insbesondere lässt sich die interne Repräsentation refaktorisieren, ohne das äußere Interface ändern zu müssen.

Aber zurück zum eigentlichen Thema – Zero-Bonds gehen ja schon, Swaps aber noch nicht. Ein Swap lässt sich als Kombination von zwei Zero-Bonds repräsentieren, das suggeriert folgenden Kombinator:

```
type Contract =
  ...
  | Both of Contract * Contract
```

Nach einem solchen Kombinator, der aus zwei „Dingsen" wieder ein „Dings" macht, sollten wir immer suchen, denn so etwas gibt es (manchmal überraschenderweise) in nahezu jeder Domäne. Der dazugehörige Funktions-Wrapper sieht dann so aus:

```
let both contract1 contract2 =
  Both (contract1, contract2)
```

Natürlich kann das Ergebnis von *both* auch wieder in *both* hineingefüttert werden. Es ist also problemlos möglich, auch drei oder mehr Verträge zu kombinieren.

Und so würden wir weitermachen: Wir lassen uns von den Bankern noch mehr Verträge vorlegen und versuchen, diese mit den existierenden Kombinatoren abzubilden. Wenn das scheitert, dann erweitern wir den Satz Kombinatoren.

Ein Kombinator lauert noch in den beiden Zero-Bonds: Beim ersten steht da *Bekomme*, beim zweiten *Zahle*, die Richtung ist also unterschiedlich. Standard bei den bisherigen Verträgen ist immer *Bekomme*. Es fehlt also noch ein Kombinator, der die Richtung der Zahlungen umdreht:

```
type Contract =
  ...
  | Give of Contract
```

Einen weiteren Konstruktor könnte man aus ganz prinzipiellen Erwägungen noch hinzufügen: Immer wenn es einen zweistelligen Kombinator wie *Both* gibt, sollten wir fragen, ob wir damit nicht einen Monoid bilden können. Monoid ist ein Konzept aus der abstrakten Algebra, von wo eine ganze Reihe nützlicher Ideen für die funktionale Programmierung stammen. Ein Monoid ist eine Struktur mit folgenden Eigenschaften:

- Es gibt eine Menge von Werten – in diesem Fall der Typ *Contract* –,
- mit einem zweistelligen Kombinator wie *Both*,
- auf der das Assoziativgesetz gilt (mehr dazu später),
- und es gibt ein neutrales Element.

Ein neutrales Element ist ein Wert, der keine Auswirkung hat, wenn er mit einem anderen Wert kombiniert wird. Zum Beispiel ist bei Zahlen 0 das neutrale Element der Addition, und das neutrale Element der Multiplikation ist die 1.

In Bezug auf *Both* wäre dafür ein „leerer" Vertrag erforderlich, bei dem überhaupt keine Zahlungen erfolgen. Wir fügen ihn einfach hinzu und werden später noch sehen, wozu er gut ist, auch wenn es in der Praxis keine leeren Verträge gibt. Damit sieht *Contract* jetzt so aus:

```
type Contract =
  | Zero
  | One of Currency
```

```
  | Multiple of Amount * Contract
  | Later of Date * Contract
  | Both of Contract * Contract
  | Give of Contract
```

Für *Zero* und *Give* werden außerdem noch Wrapper-Funktionen definiert:

```
let zero = Zero
let give contract = Give contract
```

Würden wir jetzt noch weitere Verträge aus der Banker-Praxis analysieren, kämen dabei noch folgende weitere Kombinatoren heraus, deren Code aus Platzgründen hier weglassen wird (Details finden Sie unter [2]):

- *Or*, das Gegenstück von *And*, bei dem der Inhaber sich aussuchen kann, welchen der beiden Teilverträge er nimmt.
- *Cond*, der einen Vertrag abhängig davon auswählt, ob eine bestimmte Bedingung eintritt.
- *Anytime*, das einen Vertrag verzögert wie *Later*, bei dem sich aber der Inhaber aussuchen kann, wann er eintritt.
- *Until*, der einen Vertrag storniert, wenn eine Bedingung eintritt.

Und das war's: Zehn Kombinatoren, die so gut wie alle bekannten Finanzderivate abbilden.

Dem vereinfachten Modell in diesem Artikel fehlt noch ein Aspekt: Viele der Größen – wie zum Beispiel die Menge bei *Multiple* oder der Zeitpunkt bei *Later* – müssen zeitabhängig sein. Details dazu, wie das geht – auch mit einem Kombinator-Modell – finden Sie im Fachaufsatz unter [2]. Aus dem dortigen Domänenmodell sind ein Produkt und eine Firma (LexiFi in Frankreich) entstanden, welche die Validität des Modells bestätigt.

Zusammenfassend ist es also gar nicht notwendig, jeden Vertrag einzeln und von Neuem abzubilden: Stattdessen wird er aus einem festen Satz an Bausteinen zusammengebaut. Der F#-Code für das Modell hat gerade einmal elf Zeilen.

Das Modell könnte man auch in C# abbilden – ein Interface und zehn Klassen, die es implementieren. Leider ist der Code dafür um ein Vielfaches größer. Das wäre für sich genommen noch kein so großer Unterschied, in der Praxis macht C# es aber deutlich schwerer, solche flexiblen Modelle zu denken. Dementsprechend selten existieren sie in der objektorientierten Welt.

Semantik von Verträgen

Noch größere Vorteile spielt F# aus, wenn es darum geht, zu definieren, was die Verträge eigentlich bedeuten. Bisher wurde ja bloß eine Repräsentation definiert mit mehr oder weniger verständlicher Erläuterung, wofür die Vertrags-Objekte stehen. Um einem System mit dieser Repräsentation aber Millionen von Euro anzuvertrauen, sollten man präzise definieren, was so ein Vertrag eigentlich ist – eine Semantik.

Unterschiedliche Abteilungen einer Bank sehen unterschiedliche Arten von Semantik: Das Front Office sieht eher den Wert eines Vertrags, während das Back Office die Zahlungen veranlassen muss, die sich aus einem Vertrag ergeben. Hier machen wir uns die Back-Office-Sicht zu eigen, die definiert, welche Zahlungen ein Vertrag veranlasst.

Im Back Office (so die vereinfachte Idee) schaut jemand jeden Tag auf den Vertrag und ermittelt, ob Zahlungen fällig sind. Diese werden veranlasst – außerdem muss protokolliert werden, was vom Vertrag noch „übrig" ist. Erforderlich ist also eine Funktion, die so anfängt:

```
let rec step (contract: Contract)
  (date: Date): list<Payout> * ... =
```

Das *rec* besagt, dass die Funktion – wie sich noch ergeben wird – rekursiv ist. Als Eingabe dienen also der Vertrag und das aktuelle Datum, heraus kommt eine Liste von Zahlungen (die Definition von *Payout* folgt gleich noch).

Aber da fehlt noch etwas, nämlich was übrig bleibt vom Vertrag. Man könnte versuchen, einen Typ für den „Zustand" des Vertrags zu definieren. Viel effektiver ist das Prinzip der funktionalen Programmierung, möglichst wenige Typen zu definieren. Letztlich ist das eine direkte Schlussfolgerung aus dem Prinzip, Operationen zu definieren, die den gleichen Typ für Ein- und Ausgabe verwenden.

Der Typ *Contract* ist ja explizit dafür gemacht, alle Finanzverträge zu definieren. Er sollte deshalb auch in der Lage sein, den „Rest" eines Vertrags zu repräsentieren. Die Typ-Signatur von *step* wird deshalb wie folgt ergänzt:

```
let rec step (contract: Contract)
  (date: Date): list<Payout> * Contract =
```

Bevor es weitergeht, fehlt noch eine Definition für den Typ für Zahlungen, *Payout*. Der sollte festlegen, wann wie viel von welcher Währung und in welche Richtung gezahlt wird:

```
type Direction = Long | Short
type Payout =
  Payout of Date * Direction * Amount * Currency
```

Long ist übrigens Banker-Sprech für *ich bekomme* und *Short* für *ich zahle*.

Um *step* zu definieren, gilt es zwischen den verschiedenen Konstruktoren von *Contract* zu unterscheiden. Dazu wird Pattern Matching verwendet, eine sehr kompakte Notation, um gleichzeitig nach Konstruktor zu verzweigen und die entsprechenden Daten zu dekonstruieren:

```
match contract with
| Zero -> ...
| One currency -> ...
| Multiple (amount', contract') -> ...
| Later (date', contract') -> ...
| Both (contract1, contract2) -> ...
| Give contract' -> ...
```

Man könnte jetzt für jeden Zweig hinschreiben, was bei *step* jeweils herauskommen soll. Außerdem bekommen die At- ▶

tribute der einzelnen Konstruktoren jeweils Namen. Bei *One* zum Beispiel bekommt die Währung den Namen *currency*, bei *Later* bekommt das Datum den Namen *date'* (der Apostroph gehört hier zum Namen, denn *date* ist ja das Datum von heute) und der Vertrag, der verzögert wird, bekommt den Namen *contract'*.

Nun lassen sich die einzelnen Zweige wie ein Formular ausfüllen – wir müssen uns für jeden überlegen, was für Zahlungen herauskommen und was vom Vertrag danach übrig bleibt, hier *Zero* und *One*:

```
| Zero -> ([], zero)
| One currency ->
  ([Payout (date, Long, 1.0, currency)], zero)
```

Bei *Zero* (der „leere Vertrag“) kommt keine Zahlung heraus und es bleibt auch nichts übrig. Bei *One* kommt eine einzelne Zahlung – ein Stück Währung – heraus und nichts bleibt übrig (und bei der Gelegenheit sehen Sie auch, wozu *Zero* gut ist). Spannender wird es bei *Multiple*:

```
| Multiple (amount', contract') ->
  let (payouts', contract_res') = step contract' date
  (List.map (scale_payout amount') payouts',
  multiple amount' contract_res')
```

Da in *Multiple* wieder ein Vertrag steckt, ist ein rekursiver Aufruf erforderlich, der besagt, welche Zahlungen der unskalierte Vertrag *contract'* liefern würde und was davon übrig bliebe. Diese Zahlungen gilt es zu skalieren, genauso wie den Rest-Vertrag *contract_res'*. Für Letzteren kommt wieder *multiple* zum Einsatz, für Ersteres fehlt noch eine Funktion, welche eine Zahlung skaliert. Auch das klappt dank Pattern-Matching:

```
let scale_payout (factor: double)
  (Payout (date, direction, amount, currency)): Payout =
  Payout (date, direction, factor * amount, currency)
```

Die Funktion *List.map* erweitert schließlich die Funktion *scale_payout*, die auf einer einzelnen Zahlung operiert, auf eine ganze Liste davon.

Bei *Later* wird überprüft, ob das Datum der Verzögerung schon abgelaufen ist. Wenn ja, wird *step* auf den verzögerten Vertrag rekursiv angewendet. Sonst wird keine Zahlung gemeldet und der Vertrag unverändert zurückgeliefert:

```
| Later (date', contract') ->
  if date >= date'
  then step contract' date
  else ([], contract)
```

Man sieht hier ein weiteres kleines, aber feines Feature von F#, das den Vergleich von zwei *Date*-Objekten automatisch definiert.

Bei *Both* muss *step* sich auf beiden Teilverträgen rekursiv aufrufen und die Ergebnisse wieder kombinieren:

```
| Both (contract1, contract2) ->
  let (payments1, contract1_res) = step contract1 date
  let (payments2, contract2_res) = step contract2 date
  (List.append payments1 payments2,
  both contract1_res contract2_res)
```

Bei *Give* schließlich ist ebenfalls ein rekursiver Aufruf fällig, dessen Ergebnis aber wieder umgedreht werden muss. Dafür wird noch die Hilfsfunktion *invert_payout* benötigt:

```
  | Give contract' ->
    let (payments', contract_res') = step contract' date
    (List.map invert_payout payments',
    give contract_res')

let invert_direction direction =
  match direction with
  | Long -> Short
  | Short -> Long

let invert_payout (Payout (date, direction,
  amount, currency)) = Payout (date,
  invert_direction direction, amount, currency)
```

Fertig ist die Semantik! Und das Tolle ist – sie funktioniert für alle Verträge und muss nicht bei jedem neuen Produkt definiert werden, wie es leider in vielen „Portfolio-Management-Systemen“ aus der Banken-Praxis üblich ist.

Natürlich lässt sich auch die Funktion *step* objektorientiert in C# abbilden. Aber die Idee, den Zustand eines Vertrags wieder durch einen Vertrag abzubilden, ist der objektorientierten Programmierung eher fremd – ein typischer objektorientierter Ansatz wäre deutlich komplizierter geworden. Die kompakte Schreibweise von F# erlaubt außerdem, dass wir uns auf das Wesentliche konzentrieren, ohne den Überblick zu verlieren.

Smart Constructors und das Assoziativgesetz

Ein Schönheitsfehler bleibt noch. Er fällt besonders dann auf, wenn man *step* mithilfe der interaktiven Shell für F# (F# Interactive [4]) ausprobiert:

```
step zcb1 (Date "2020-12-24");;
 val it : Payout list * Contract =
   ([Payout (Date "2020-12-24",Long,100.0,EUR)],
   Multiple (100.0,Zero))
```

Der Rest-Vertrag ist ein Vielfaches von *Zero*, gemeinhin also gleichbedeutend mit *Zero* – das *Multiple* ist aber stehengeblieben: Das ist nur schwer zu sehen und man kann sich noch deutlich kompliziertere Versionen von „Nichts“ vorstellen. Es wäre schöner, wenn da einfach *Zero* stünde. Außerdem wurde ja postuliert, dass *Zero* das neutrale Element bezüglich *Both* werden sollte, aber hier bleiben *Both* und *Zero* stehen:

```
both zcb1 Zero;;
 val it : Contract =
```

```
   Both (Later (Date "2020-12-24",
   Multiple (100.0,One EUR)),Zero)
```

Das lässt sich vermeiden, indem man die Wrapper-Funktionen für die Konstruktoren zu sogenannten „Smart Constructors" macht, die erkennen, dass sich der entstehende Vertrag vereinfachen lässt. Auch das klappt wunderbar mit Pattern-Matching:

```
let rec multiple amount contract =
  match contract with
  | Zero -> Zero
  | Multiple (amount', contract') ->
    multiple (amount * amount') contract'
  | _ -> Multiple (amount, contract)
```

Der erste Zweig macht aus *Multiple (..., Zero)* einfach nur *Zero*. Interessanter ist vielleicht der zweite, der aus zwei ineinandergeschachtelten *Multiple* ein einzelnes macht. Erst der letzte Zweig ruft dann den Konstruktor aus. Ähnlich kann man das auch mit den anderen Wrappern machen:

```
let give contract =
  match contract with
  | Zero -> Zero
  | Give contract' -> contract'
  | _ -> Give contract

let later date contract =
  match contract with
  | Zero -> Zero
  | _ -> Later (date, contract)

let both contract1 contract2 =
  match (contract1, contract2) with
  | (Zero, _) -> contract2
  | (_, Zero) -> contract1
  | _ -> Both (contract1, contract2)
```

Bei *both* sieht man, dass man Pattern-Matching auch verwenden kann, um zwei Werte im Zusammenhang miteinander zu analysieren.

Weitere Optimierungen in den Smart Constructors sind möglich. Oben wurde zum Beispiel noch das Assoziativgesetz angesprochen, das man aus der Schule kennt, zum Beispiel als folgende Gleichung für beliebige Zahlen *a*, *b* und *c*:

```
a + (b + c) = (a + b) + c
```

Übertragen auf beliebige Verträge mit den Namen *a*, *b* und *c* heißt das:

```
both a (both b c) = both (both a b) c
// Das gilt bisher noch nicht:
both (one EUR) (both (one GBP) (one EUR));;
 val it : Contract =
 Both (One EUR,Both (One GBP,One EUR))
both (both (one EUR) (one GBP)) (one EUR);;
  val it : Contract =
  Both (Both (One EUR,One GBP),One EUR)
```

Wir können das nachholen, indem das Pattern-Matching in *both* so erweitert wird, dass es geschachtelte *both*-Aufrufe nach rechts umklammert:

```
let rec both contract1 contract2 =
  match (contract1, contract2) with
  | (Zero, _) -> contract2
  | (_, Zero) -> contract1
  | (Both (a, b) , c) -> both a (both b c)
  | _ -> Both (contract1, contract2)
```

Noch einmal: Prinzipiell geht das auch in C#, aber es ist dort erheblich umständlicher. F# ist so kompakt, dass wir es nicht nur als Implementierungssprache, sondern bei der Domänenmodellierung auch sehr gut als Werkzeug zum Denken benutzen können.

Fazit

F# ist C# in so gut wie jeder Hinsicht überlegen. Das zeigt sich besonders in der Domänenmodellierung, wo F# schneller zu besseren Modellen führt. Lediglich bei der Verbreitung liegt F# noch deutlich hinter C#.

Das muss aber niemanden abhalten, es in einem Projekt einzusetzen, auch wenn die Unterschiede zur objektorientierten Programmierung groß sind: Funktionale Programmierung lässt sich einfach erlernen (beispielsweise mit der Einführung in die funktionale Programmierung unter [5]), und F# ist gegenüber C# die deutlich kleinere und weniger komplexe Sprache. Probieren Sie's aus! ■

[1] Domain-Driven Design bei Wikipedia, www.dotnetpro.de/SL2012FSharp1

[2] S.L. Peyton Jones, J-M. Eber, How to write a financial contract. In „The Fun of Programming", edited by J. Gibbons and O. de Moor, Palgrave Macmillan 2003, ISBN 978-0-333-99285-2

[3] Eric Evans, Domain-Driven Design, Addison-Wesley, 2003, ISBN 978-0-321-12521-7

[4] F# Interactive, www.dotnetpro.de/SL2012FSharp2

[5] M. Sperber, H. Klaeren, Schreibe Dein Programm (PDF), Einführung in die funktionale Programmierung, www.deinprogramm.de

Dr. Michael Sperber
ist Spezialist für die funktionale Programmierung sowie Autor zahlreicher Fachveröffentlichungen zu diesem Thema. Er hat jahrelange Erfahrung in der Softwareentwicklung, ist Experte für die Programmierausbildung und Geschäftsführer der Active Group GmbH.

dnpCode A2012FSharp